

A DIFFERENTIABLE PHYSICS ENGINE FOR DEEP LEARNING IN ROBOTICS

Jonas Degraeve, Michiel Hermans*, Joni Dambre & Francis wyffels

Department of Electronics and Information Systems (ELIS)

Ghent University – iMinds, IDLab

Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium

{Jonas.Degraeve, Joni.Dambre, Francis.wyffels}@UGent.be

ABSTRACT

One of the most important fields in robotics is the optimization of controllers. Currently, robots are treated as a black box in this optimization process, which is the reason why derivative-free optimization methods such as evolutionary algorithms or reinforcement learning are omnipresent. We propose an implementation of a modern physics engine, which has the ability to differentiate control parameters. This has been implemented on both CPU and GPU. We show how this speeds up the optimization process, even for small problems, and why it will scale to bigger problems. We explain why this is an alternative approach to deep Q-learning, for using deep learning in robotics. Lastly, we argue that this is a big step for deep learning in robotics, as it opens up new possibilities to optimize robots, both in hardware and software.

1 INTRODUCTION

In order to solve tasks efficiently, robots require an optimization of their control system. This optimization process can be done in automated testbeds, but typically these controllers are optimized in simulation. Common methods to optimize these controllers include particle swarms, reinforcement learning, genetic algorithms and evolutionary strategies. These are all derivative-free methods.

However, deep learning has taught us that optimizing with a gradient is often faster and more efficient. This is especially true when there are a lot of parameters, as is common in deep learning. However, in these optimization processes the robot is almost exclusively treated as a non differentiable black box. The reason for this, is that the robot in hardware is not differentiable, nor are current physics engines able to provide the gradient of the robot models. The resulting need for derivative-free optimization approaches limits both the optimization speed and the number of parameters in the controllers.

A recently popular approach is to use deep Q-learning, a reinforcement learning algorithm. This method requires a lot of evaluations in order to work and to learn the many parameters (Levine et al., 2016). Here, we suggest an alternative approach, by introducing a differentiable physics engine. This idea is not novel. It has been done before with spring-damper models in 2D and 3D (Hermans et al., 2014). This approach is also similar to adjoint optimisation, a method widely used in various applications such as thermodynamics (Jarny et al., 1991) and fluid dynamics (Iollo et al., 2001). However, modern engines to model robotics are based on different algorithms. The most commonly used ones are 3D rigid body engines, which rely on impulse-based velocity stepping methods (Erez et al., 2015). In this paper, we test whether these engines are also differentiable, and whether this gradient is computationally tractable.

*Former member, currently unaffiliated

2 A 3D RIGID BODY ENGINE

The goal is to implement a modern 3D Rigid body engine, in which parameters can be differentiated with respect to the fitness a robot achieves in a simulation, such that these parameters can be optimized with methods based on gradient descent.

The most frequently used simulation tools for model-based robotics, such as PhysX, Bullet, Havok and ODE, go back to MathEngine (Erez et al., 2015). These tools are all 3D rigid body engines, where bodies have 6 degrees of freedom, and the relations between them are defined as constraints. These bodies exert impulses on each other, but their positions are constrained, e.g. to prevent the bodies from penetrating each other. The velocities, positions and constraints of the rigid bodies define a linear complementarity problem (LCP) (Chappuis, 2013), which is then solved using a Gauss-Seidel projection (GSP) method (Jourdan et al., 1998). The solution of this problem are the new velocities of the bodies, which are then integrated by semi-implicit Euler integration to get the new positions (Stewart and Trinkle, 2000). This system is not always numerically stable, therefore the constraints are usually softened (Catto, 2009).

We implemented such an engine as a mathematical expression in Theano (Al-Rfou et al., 2016), a software library which does automatic evaluation and differentiation of expressions with a focus on deep learning. This library has allowed for efficient differentiation of remarkably complex functions before (Degraeve et al., 2016). The resulting computational graph to evaluate this expression, is then compiled for both CPU and GPU. In order to be able to compile for GPU however, we had to limit our implementation to a restricted set of elementary operations. This has some drawbacks, such as a limited support for conditionals. This severely caps the range of implementable functions. However, since the gradient is determined automatically, the complexity of implementing the differentiation correctly is removed entirely.

One of these limitations, is that we need to implement our physics engine without branching, as this is not yet available in Theano for GPU. Therefore some sacrifices have to be made. For instance, our system only allows for contact constraints between different spheres or between spheres and the ground plane. Collision detection algorithms for cubes typically have a lot of branching (Mirtich, 1998). However, this sphere based approach can in principle be extended to any other shape (Hubbard, 1996). On the other hand, we did implement a rather accurate model of servo motors, with a gain, maximal torque and maximal velocity parameters.

Another design choice was to use rotation matrices rather than the more common quaternions for representing rotations. This means that the state of the object is bigger, but the operations required are matrix multiplications. This reduced the complexity of the graph. However, cumulative operations on a rotation matrix might move the matrix away from orthogonality. To correct for this, we renormalize our matrix with the update equation (Premierani and Bizard, 2009):

$$A' = \frac{3A - A \circ (A \cdot A)}{2} \quad (1)$$

These design decisions are the most important aspects of difference with the frequently used simulation tools. In the following section, we will evaluate our physics simulator on a number of different problems. We take a look at the speed of computation, and the number of evaluations required before the parameters of are optimized.

3 RESULTS

3.1 THROWING A BALL

To test our engine, we implemented the model of a giant soccer ball in the physics engine, as shown in Fig. 1a. The ball has a 1 m diameter, a friction of $\mu = 1.0$ and restitution $e = 0.5$. The ball starts off at position $(0, 0)$. After 5 s it should be at position $(10, 0)$ with zero velocity v and zero angular velocity ω . We optimized the initial velocity v_0 and angular velocity ω_0 at time $t = 0$ s until the errors at $t = 5$ s are less than 0.01 m and 0.01 m/s, respectively.

Since the quantity we optimize is only known at the end of the simulation, but we need to optimize the parameters in the beginning of the simulation, we need to backpropagate our error through time

(BPTT) (Sutskever, 2013). This is similar to the backpropagation through time method used for optimizing recurrent neural networks (RNN). In our case, every time step in the simulation can be seen as one pass through a neural network, which transforms the inputs from this timestep to inputs for the next time step. For finding the gradient, this RNN is unfolded completely, and the gradient can be found by differentiation this unfolded structure. This differentiation is done automatically by the Theano library.

Optimizing the 6 parameters in v_0 and ω_0 took only 88 iterations with gradient descent and back-propagation through time. Optimizing this problem with CMA-ES (Hansen, 2006), a state of the art derivative-free optimization method, took 2422 iterations. Even when taking the time to compute the gradient into account, the optimization with gradient descent takes 16.3s, compared to 59.9s with CMA-ES. This shows that gradient based optimization of kinematic systems can in some cases already outperform general purpose optimization algorithms from as little as six parameters.

3.2 QUADRUPEDAL ROBOT

To verify the speed of our engine, we also implemented a small quadrupedal robot model, as illustrated in Fig. 1b. This model has a total of 81 sensors, e.g. encoders and an inertial measurement unit (IMU). The servo motors are controlled in closed loop by a small neural network with a varying number of parameters, as shown in Fig. 1c. The gradient is the Jacobian of the total travelled distance of the robot in 10s, differentiated with respect to all the parameters of the controller. This Jacobian is found by using BPTT and propagating all 10s back. The time it takes to compute this travelled distance and the accompanying Jacobian is shown in Table 1. We include both the computation time with and without the gradient, i.e. both the forward and backward pass and the forwards pass alone. This way, the numbers can be compared to other physics engines, as they only calculate without gradient. Our implementation and our model can probably be made more efficient, and evaluating the gradient can probably be made faster a similar factor.

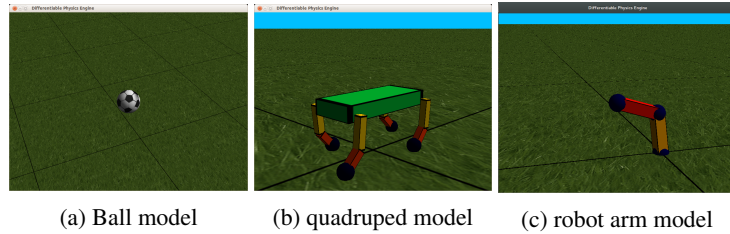


Figure 1: (a) Illustration of the ball model used in the first task. (b) Illustration of the quadruped robot model with 8 actuated degrees of freedom, 1 in each shoulder, 1 in each elbow. The spine of the robot can collide with the ground, through 4 spheres in the inside of the cuboid. (c) Illustration of the robot arm model with 4 actuated degrees of freedom.

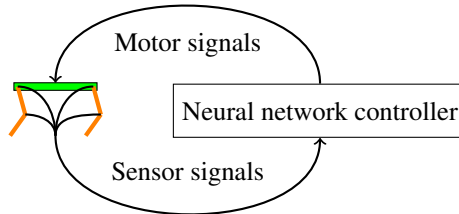


Figure 2: Illustration of the closed loop controller. The neural network receives sensor signals from the encoders on the joints, and uses these to generate motor signals which are sent to the servo motors.

When only a single controller is optimized, our engine runs more slowly on GPU than on CPU. In order to tackle this issue, we implemented batch gradient descent, which is commonly used in

Table 1: Evaluation of the computing speed of our engine on a robot model controlled by a closed loop controller with a variable number of parameters. We evaluated both on CPU (i7 5930K) and GPU (GTX 1080), both for a single robot optimization and for batches of multiple robots in parallel. The numbers are the time required in seconds for simulating the quadruped robot(s) for 10 s, with and without calculating a gradient. The gradient calculated here is the Jacobian of the total travelled distance of the robot in 10 s, differentiated with respect to all the parameters of the controller.

Seconds of computing time required to simulate a batch of robots for 10 seconds

		with gradient		without gradient	
		CPU	GPU	CPU	GPU
1 robot	1 296 parameters	8.17	69.6	1.06	9.69
	1 147 904 parameters	13.2	75.0	2.04	9.69
128 robots	1 296 parameters	263	128	47.7	17.8
	1 147 904 parameters	311	129	50.4	18.3

complex optimization problems. In this case, by batching our robot models, we achieve considerable acceleration on GPU. Although backpropagating the gradient through time slows down the computations by roughly a factor 10, this factor only barely increases with the number of parameters in our controller. Combining this with our previous observation that fewer iterations are needed when using gradient descent, our approach can enable the use of gradient descent through physics for highly complex deep learning controllers with millions of parameters. Also note that by using a batch method, a single GPU can simulate 864 000 model seconds per day. This should be plenty for deep learning. It also means that a single simulation step of a single robot, which includes collision detection, solving the LCP problem, integrating the velocities and backpropagating the gradient through it all, takes about 1 ms on average. Without the backpropagation, this is only about seven times faster.

3.3 4 DEGREE OF FREEDOM ROBOT ARM

As a first test of optimizing robot controllers, we implemented a four degree of freedom robotic arm, as depicted in Fig. 1c. The bottom of the robot has a 2 degrees of freedom actuated universal joint, the elbow has a 2 degree of freedom actuated joint as well. The arm is 1 m long, and has a total mass of 32 kg. The servos have a gain of 30 s^{-1} , a torque of 30 Nm and a velocity of 45° s^{-1} .

For this robot arm, we train controllers for a task with a decreasing amount of difficulty. In order to be able to train our parameters, we have to use a couple of tricks often used in the training of recurrent neural networks.

- We choose an objective which is evaluated at every time step and then averaged, rather than at specific points of the simulation. This vastly increases the amount of samples over which the gradient is averaged, which in turn makes the gradient direction more reliable (Sjöberg et al., 1995).
- The value of the gradient is decreased by a factor $\alpha < 1$ at every time step. This has the effect of a prior: namely events further in the past are less important for influencing current events, because intermediate events might diminish their influence completely. This also improves robustness against exploding gradients (Hermans et al., 2014).
- We initialize the controller intelligently. We do not want the controller to shake the actuators violently and explore outside the accurate domain of our simulation model, therefore controllers are initialized such that they only output zeros at the start of the simulation.
- We constraint the size of the gradient to an L2-norm of 1. This makes sure that gradients close to discontinuities in the fitness landscape do not push the parameter values too far away, such that everything which was learnt is forgotten (Sutskever, 2013).

3.3.1 REACHING A FIXED POINT

A first simple task, is to have a small neural net controller learn to move the controller to a certain fixed point in space, at coordinates (0.5 m; 0.5 m; 0.5 m). The objective we minimize for this task, is the distance between the end effector and the target point, averaged over the 8 seconds we simulate our model.

We provide the controller with a single sensor input, namely the current distance between the end effector and the target point. Input is not required for this task, as there are solutions for which the motor signals are constant in time. However, this would not necessarily be the optimal approach for minimizing the average distance over time, it only solves the distance at the end of the simulation, but does not minimize the distance during the trajectory to get at the final position.

As a controller, we use a dense neural network with 1 input, 2 hidden layers of 128 units with a rectifier activation function, and 4 outputs with an identity activation function. This controller has 17 284 parameters in total.

We use gradient descent with a batch size of 1 robot for optimization, as the problem is not stochastic in nature. The parameters are optimized with Adam’s rule (Kingma and Ba, 2014) with a learning rate of 0.001. Every update step with this method takes about 5 seconds on CPU. We find that the controller comes within 4 cm of the target in 100 update steps, and within 1 cm in 150 update steps, which is quite small compared to the 1 m arm of the robot. Moreover, the controller does find a more optimal trajectory which takes into account the sensor information. Solving problems like these, in less iteration steps than the number of parameters, is completely unfeasible with derivative free methods (Sjöberg et al., 1995).

Despite that, we did try to optimize the same problem with CMA-ES. Note that the target volume is 500 000 times smaller than the reachable volume of the robot, therefore from a rough estimation about 1 in 500 000 random controllers should function. After 15 000 iterations, CMA-ES did not manage to lower the average error below the error of randomly sampled controllers.

3.3.2 REACHING A RANDOM POINT

As a second task, we sample a random target point in space, inside the cuboid between (1 m; 1 m; 1 m) and (−1 m; −1 m; 0 m), parallel to the axes. We give this point as input to the controller, and the task is to again minimize the average distance between the end effector and the target point.

As a controller, we use the same dense neural network as in the previous section, but this time with 3 inputs. This controller has 17 540 parameters in total. To train for this task, we use a batch size of 64 robots, such that every update step takes 52 s on GPU. Each simulation takes 8 s with a simulation step of 0.01 s, therefore the gradient on the parameters of the controllers are averaged over 51 200 timesteps at every update step.

We find that it takes 5 000 update steps before the 17 540 parameters are optimized, such that the robot has on average 10 cm accuracy. About half of the reachable space can be reached within 5 cm accuracy.

3.4 OPTIMIZING A CONTROLLER FOR THE QUADRUPEDAL ROBOT – REVISITED

Optimizing a gait for a quadrupedal robot is a problem of a different order, something the authors have extensive experience with (Sproewitz et al., 2013; Degraeve et al., 2013; 2015). The problem is way more difficult, and allows for a wide range of possible solutions. In nature, we find a wide range of gaits, from hopping over trotting, walking and galloping. With hand tuning, we were able to obtain a trotting gait on this robot model, with an average forward speed of 0.7 m/s. We found it tricky to find a gait where the robot did not end up as an upside down turtle, as 75% of the mass of the robot is located in its spine.

As a controller for our quadrupedal robot, we use a neural network with 2 input signals, namely a sine and a cosine signal with a frequency of 1.5 Hz. On top of this we added 2 hidden layers of 128 units and a rectifier activation function. As output layer, we have a dense layer with 8 units and a linear activation function, which has as input both the input layer and the top layer of the hidden

layers. In total, this controller has 17 952 parameters. Since the problem is not stochastic in nature, we use a batch size of 1 robot. We initialize the output layer with zero weights, so the robot starts the optimization in a stand still position.

We optimize these parameters to maximize the average velocity of the spine over the course of 10 s of time in simulation. This way, the gradient used in the update step is effectively an average of 1000 time steps after unrolling the recurrent connections. This objective does not take into account energy use, or other metrics normally used in robotic problems.

In only 500 iterations, or about 1 hour of optimizing on CPU, the optimization through BPTT comes up with a solution with a speed of 1.17 m/s. This solution is a hopping gait, with a summersault every 3 steps, despite limiting the torque of the servos to 4 Nm on this 28.7 kg robot. For more life-like gaits, energy use could be used as a regularization method. This is however outside of the scope of this paper.

4 DISCUSSION

Our results show the first prototype of a differentiable physics engine based on similar algorithms as those that are commonly used in current robotics simulators. When originally addressing the problem, we had no idea whether finding the gradient would be computationally tractable, let alone whether evaluating it would be fast enough to be beneficial for optimization. We have now demonstrated that evaluating the gradient is expensive, but on a very manageable level. The speed of evaluating the gradient mainly depends on the complexity of the physics model and only slightly on the number of parameters to optimize. Our results therefore suggest that this cost can be dominated by the gain that can be achieved by the combination of using batch gradient descent and GPU acceleration. This is especially true when optimizing controllers with very high numbers of parameters, where we suspect this approach is asymptotically of a lower order in the number of parameters, as each gradient step also contains information proportional to the number of parameters.

Optimizing the controller of a robot model with gradient based optimization is equivalent to optimizing a RNN. After all, the gradient passes through each parameter at every time step. The parameter space is therefore very noisy. Consequently, training the parameters of this controller is a highly non-trivial problem, as it corresponds to training the parameters of a RNN. On top of that, exploding and vanishing signals and gradients cause far more difficult problems compared to feed forward networks.

In section 3.3, we already discussed some of the tricks used for optimizing RNNs. Earlier research shows that these methods can be extended to even more difficult tasks than the ones discussed here (Hermans et al., 2014; Sutskever, 2013). We therefore believe that this approach towards learning controllers for robotics is applicable to far more complex problems than the simple examples tackled in this paper.

5 FUTURE WORK

All of the results in this paper will of course largely depend on showing how these controllers will work on the physical counterparts of our models. Nonetheless, we would like to conjecture that to a certain extent, this gradient of a model is close to the gradient of the physical system. The gradient of the model is even more susceptible to high-frequency noise introduced by modeling the system, than the imaginary gradient of the system itself. Nonetheless, it contains information which might be indicative, even if it is not perfect. We would theorize that using this noisy gradient is still better than optimizing in the blind, and that the transferability to real robots can be improved by evaluating the gradients on batches of (slightly) different robots in (slightly) different situations and averaging the results. This technique was already applied in (Hermans et al., 2014) as a regularization technique to avoid bifurcations during online learning. If the previous proves to be correct, our approach can offer an alternative to deep Q-learning for deep learning controllers in robotics.

We can see the use of this extended approach for a wide range of applications in robotics:

- By introducing recurrent connections in the neural network controller, it is possible to introduce memory in the controller as well. We reckon that advanced recurrent connections

with a memory made out of LSTM cells (Hochreiter and Schmidhuber, 1997) can allow for more powerful controllers than the controllers described in this paper.

- Although we did not address this in this paper, there is no reason why only control parameters could be differentiated. Hardware parameters of the robot have been optimized the same way before (Jarny et al., 1991; Iollo et al., 2001; Hermans et al., 2014).
- There is no reason why a camera model would not be differentiable either. We think that currently the only thing in the way to effectively learn deep robot controllers from camera information, is someone implementing a differentiable camera model.
- Where adversarial networks are already showing their use in generating image models, we believe adversarial robotics training (ART) will create some imaginative ways to design and control robots. Like in generative adversarial nets (GAN) (Goodfellow et al., 2014), where the gradient is pulled through two competing neural networks, the gradient could be pulled through multiple competing robots as well.

6 CONCLUSION

In this paper, we show it is possible to build a differentiable physics engine. We implemented a modern engine which can run a 3D rigid body model, using the same algorithm as other engines commonly used to simulate robots, but we can additionally differentiate control parameters with BPTT. Our implementation also runs on GPU, and we show that using GPUs to simulate the physics can speed up the process for large batches of robots.

We find that these gradients can be computed surprisingly fast. We also show that using gradient descent with BPTT speeds up optimization processes often found in robotics, even for rather small problems, due to the reduced number of model evaluations required. This improvement in speed will scale to problems with a lot of parameters. This method should therefore allow for a new way to apply deep learning methods in robotics.

ACKNOWLEDGMENTS

Special thanks to Iryna Korshunova for valuable discussions and proofreading the paper. The research leading to these results has received funding from the Agency for Innovation by Science and Technology in Flanders (IWT). The GTX 1080 used for this research was donated by the NVIDIA Corporation.

REFERENCES

- Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., et al. (2016). Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*.
- Catto, E. (2009). Modeling and solving constraints. In *Game Developers Conference*.
- Chappuis, D. (2013). Constraints derivation for rigid body simulation in 3D.
- Degrave, J., Burm, M., Kindermans, P.-J., Dambre, J., et al. (2015). Transfer learning of gaits on a quadrupedal robot. *Adaptive Behavior*, page 1059712314563620.
- Degrave, J., Burm, M., Waegeman, T., Wyffels, F., and Schrauwen, B. (2013). Comparing trotting and turning strategies on the quadrupedal oncilla robot. In *Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference on*, pages 228–233. IEEE.
- Degrave, J., Dieleman, S., Dambre, J., et al. (2016). Spatial chirp-Z transformer networks. In *European Symposium on Artificial Neural Networks (ESANN)*.
- Erez, T., Tassa, Y., and Todorov, E. (2015). Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *International Conference on Robotics and Automation (ICRA)*, pages 4397–4404. IEEE.
- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680.

- Hansen, N. (2006). The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer Berlin Heidelberg.
- Hermans, M., Schrauwen, B., Bienstman, P., and Dambre, J. (2014). Automated design of complex dynamic systems. *PLoS one*, 9(1):e86696.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.
- Hubbard, P. M. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics (TOG)*, 15(3):179–210.
- Iollo, A., Ferlauto, M., and Zannetti, L. (2001). An aerodynamic optimization method based on the inverse problem adjoint equations. *Journal of Computational Physics*, 173(1):87–115.
- Jarny, Y., Ozisik, M., and Bardon, J. (1991). A general optimization method using adjoint equation for solving multidimensional inverse heat conduction. *International journal of heat and mass transfer*, 34(11):2911–2919.
- Jourdan, F., Alart, P., and Jean, M. (1998). A gauss-seidel like algorithm to solve frictional contact problems. *Computer methods in applied mechanics and engineering*, 155(1):31–47.
- Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.
- Levine, S., Pastor, P., Krizhevsky, A., and Quillen, D. (2016). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *arXiv preprint arXiv:1603.02199*.
- Mirtich, B. (1998). V-clip: Fast and robust polyhedral collision detection. *ACM Transactions On Graphics (TOG)*, 17(3):177–208.
- Premierani, W. and Bizard, P. (2009). Direction cosine matrix IMU: Theory. *DIY DRONE: USA*, pages 13–15.
- Sjöberg, J., Zhang, Q., Ljung, L., Benveniste, A., Delyon, B., Glorennec, P.-Y., Hjalmarsson, H., and Juditsky, A. (1995). Nonlinear black-box modeling in system identification: a unified overview. *Automatica*, 31(12):1691–1724.
- Sproewitz, A., Tuleu, A., D’Haene, M., Möckel, R., Degraeve, J., Vespignani, M., Gay, S., Ajallooeian, M., Schrauwen, B., and Ijspeert, A. J. (2013). Towards dynamically running quadruped robots: performance, scaling, and comparison. In *Adaptive Motion of Animals and Machines*, pages 133–135.
- Stewart, D. and Trinkle, J. C. (2000). An implicit time-stepping scheme for rigid body dynamics with coulomb friction. In *International Conference on Robotics and Automation (ICRA)*, volume 1, pages 162–169. IEEE.
- Sutskever, I. (2013). *Training recurrent neural networks*. PhD thesis, University of Toronto.